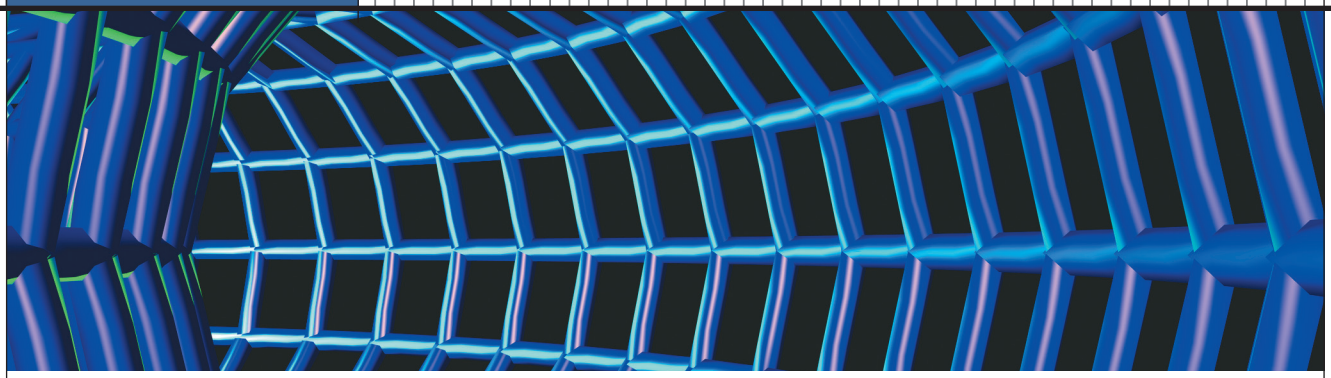
8-2011

Defending against cross site scripting attacks

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Hee Beng Kuan TAN



Defending against Cross-Site Scripting Attacks

Lwin Khin Shar and Hee Beng Kuan Tan
Nanyang Technological University, Singapore

Researchers have proposed multiple solutions to cross-site scripting, but vulnerabilities continue to exist in many Web applications due to developers' lack of understanding of the problem and their unfamiliarity with current defenses' strengths and limitations.

According to security experts, cross-site scripting is among the most serious and common threats in Web applications today, surpassing buffer overflows—the number one vulnerability for the past decade. In 2010, XSS ranked first in the Mitre Common Weakness Enumeration (CWE)/SANS Institute list of Top 25 Most Dangerous Software Errors (http://cwe.mitre.org/top25) and second in the Open Web Application Security Project (OWASP) Top 10 list of security risks (https://www.owasp.org/index.php/Top_10). Several major websites including Facebook, Twitter, Myspace, eBay, Google, and McAfee have been the targets of XSS exploits.

XSS is the result of a weakness inherent in many Web applications' security mechanisms: the absence or insufficient sanitization of user inputs. XSS flaws exist in Web applications written in various programming languages such as PHP, Java, and .NET where application webpages reference unrestricted user inputs. Attackers inject malicious code via these inputs, thereby causing unintended script executions by clients' browsers.

Researchers have proposed multiple XSS solutions ranging from simple static analysis to complex runtime protection mechanisms. However, vulnerabilities continue to exist in many Web applications due to developers' lack of understanding of the problem and their unfamiliarity with current defenses' strengths and limitations.

XSS EXPLOITS

XSS exploits are similar to SQL injection, an original form of code injection. This type of attack exploits an application's output function that references poorly sanitized user input. However, SQL injection targets the query function that interacts with the database, whereas XSS exploits target the HTML output function that sends data to the browser.

The basic idea of XSS injection is to use special characters to cause Web browser interpreters to switch from a data context to a code context.[1] For example, when an HTML page references a user input as data, an attacker might include the tag <script>, which can invoke the JavaScript interpreter. If the application does not filter such special characters, XSS injection is successful, and the attacker can perform exploits such as account hijacking, cookie poisoning, denial of service (DoS), and Web content manipulation. Typical input sources that attackers manipulate include HTML forms, cookies, URLs, and external files. Attackers often favor JavaScript, but other kinds of client-side

0018-9162/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

```
1 <html>
2 <title>Forum for Traveling Tips</title>
3 <body>
4 <h1>Welcome <script language="javascript" src="travelerInfo.js">
      </script>!</h1>

<%
5  String action = request.getParameter("Action");
6  String place = request.getParameter("Place");
7  if (place !=null && action.equals("Post")) {
8       String new_tip = request.getParameter("Tip");
9       if(new_tip.length < 100) {
10          stmt.executeUpdate("INSERT INTO forum VALUES (" +
                              place + ", "+ new_tip + ")" );
11          out.println("Your Post has been added under Place '"
                              + HTMLencode(place)+"'");
        }
        else {
12          out.println("Your Message:  '"+new_tip+ "' is too long!");
        }
   }
13 else if (place !=null && action.equals("View")) {
14      ResultSet rs = stmt.executeQuery("SELECT * FROM forum
            WHERE place= "+place);
15      out.println("Here are the tips about visiting this place...");
16      while(rs.next()) {
17           String tip = rs.getString("tip");
18           out.println("'"+tip+"'");
        }
   }
    . . .
%>
19 </body></html>

<%
String HTMLencode(String value) { //server-side escaping method
20 value.replace("&", "&amp;");
21 value.replace("<", "&lt;");
22 value.replace(">", "&gt;");
23 return value;
}
%>
```

(a)

```
<SCRIPT>
24 var pos=document.URL.indexOf("User=")+5;
25 document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```

(b)

Figure 1. Example vulnerable (a) server-side program and (b) client-side script for a Web application that lets travelers share tips about the places they have visited. The program contains four input fields—"Action," "Place," "Tip," and "User"—that attackers can manipulate.

scripts such as VBScript and Flash, which browsers can interpret, could cause XSS.

Injection methods

Depending on the ways HTML pages reference user inputs, XSS exploits can be broadly classified as *reflected*, *stored*, or *DOM-based*.

Reflected or nonpersistent XSS holes are present in a Web application server program where it references accessed user input in the outgoing webpage. This type of XSS exploit is common in error messages and search results. The XSSed project (http://xssed.com) recently reported multiple reflected XSS holes in McAfee that attackers could exploit to trick users into downloading viruses.

Stored or persistent XSS holes exist when a server program stores user input containing injected code in a persistent data store such as a database and then references it in a webpage. Attacks against social networking sites commonly exploit this type of XXS flaw. An example is the Samy worm (www.securityfocus.com/brief/18), which, within less than 24 hours after its release on 4 October 2005, caused an exponential growth of friend lists for 1 million Myspace users, effectively creating a DoS attack.

Both reflected and stored XSS holes result from improper handling of user inputs in server-side scripts. In contrast, DOM-based XSS holes appear in the Web application when client-side scripts reference user inputs, dynamically obtained from the Document Object Model structure, without proper validation. Bugzilla's bug 272620 (https://bugzilla.mozilla.org/show_bug.cgi?id=272620) is an example of a DOM-based XSS exploit.

Example XSS exploits

Figure 1a shows a snippet from a server program, travelerTip.jsp, for a Web application that lets travelers share tips about the places they have visited. The program contains four input fields—"Action," "Place," "Tip," and "User"—that attackers can manipulate. The program can be called via a URL such as the one shown in Figure 2a.

The statement at line 12 in Figure 1a is vulnerable to reflected XSS due to the replay of invalid input supplied by users (lines 8, 9, 12). An attacker could send a seemingly innocuous URL link like the one in Figure 2b to a victim via e-mail or a social networking site. The script in bold will execute on the victim's browser if the victim follows the link to travelingForum.

The statement at line 18 in Figure 1a is vulnerable to stored XSS, as the program stores user-supplied messages without proper sanitization (lines 8-10) and displays them to visitors (lines 14, 16-18). The URL in Figure 2c

```
http://travelingForum/travelerTip.jsp?Action=View&Place=Greece&User=Jesper
```
(a)

```
http://travelingForum/travelerTip.jsp?Action=Post&Place=Greece&Tip=HiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHiHi<meta%20http-equiv="refresh"%20content="0;">&User=Jesper
```
(b)

```
http://travelingForum/travelerTip.jsp?Action=Post&Place=Greece&Tip=<Script>document.location='http://hackerSite/stealCookie.jsp?cookie='+document.cookie;</Script>&User=Hacker
```
(c)

```
http://travelingForum/travelerTip.jsp?Action=View&Place=Greece&Tip=HiHi&User=Jesper<Script>document.getElementByTagName('Tip')[child].innerHTML='<b>Our Service is Bad, Please Go to Other Site!</b>'</Script>
```
(d)

Figure 2. Example URLs that direct Web users to travelingForum. Malicious scripts that cause XSS exploits are highlighted in bold. (a) Ordinary URL that activates travelerTip.jsp in "View" action. (b) URL that causes reflected XSS: it contains malicious HTML meta-script capable of making a refresh request to travelingForum's server every 0.3 seconds, potentially causing a denial of service. (c) URL that creates stored XSS scenario: it contains JavaScript capable of sending the client's cookie information to a hacker's website. (d) URL that causes DOM-based XSS: it contains script capable of injecting misleading information over an original message.

contains JavaScript capable of sending the client's cookie information to a hacker's website. Figure 3 illustrates this scenario in more detail.

The statement at line 4 in Figure 1a is vulnerable to DOM-based XSS, as the program includes a JavaScript file, travelerInfo.js, shown in Figure 1b, that accesses "User" information from the URL (line 24) and displays it, without any sanitization, to users (line 25). Similar to the reflected XSS scenario, an attacker can exploit this vulnerability using a crafted URL like that shown in Figure 2d.

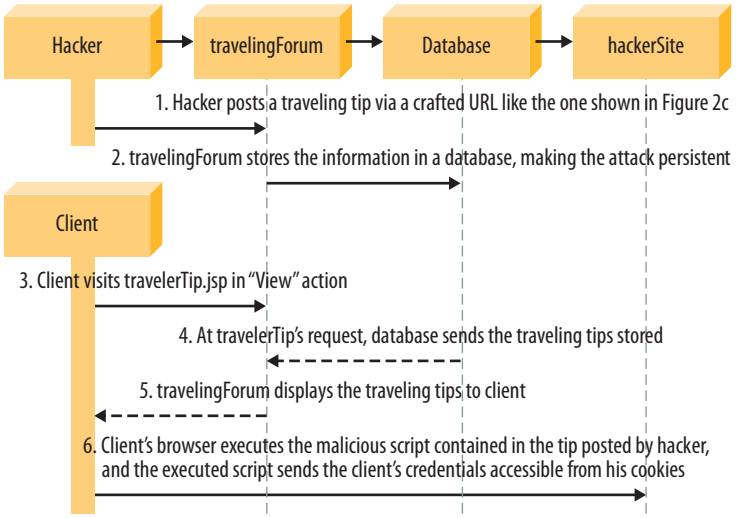


Figure 3. Stored XSS attack scenario.

XSS DEFENSES

XSS defenses can be broadly classified into four types: defensive coding practices, XSS testing, vulnerability detection, and runtime attack prevention. Table 1 compares various current techniques, which each have strengths and weaknesses.

Defensive coding

Because XSS arises from the improper handling of inputs, using defensive coding practices that validate and sanitize inputs is the best way to eliminate XSS vulnerabilities.[1,2] Input validation ensures that user inputs conform to a required input format.

There are four basic input sanitization options. *Replacement* and *removal* methods search for known bad characters (blacklist comparison); the former replaces them with nonmalicious characters, whereas the latter simply removes them. *Escaping* methods search for characters that have special meanings for client-side interpreters and remove those meanings. *Restriction* techniques limit inputs to known good inputs (whitelist comparison).

Checking blacklisted characters in the inputs is more scalable, but blacklist comparisons often fail as it is difficult to anticipate every attack signature variant. Whitelist comparisons are considered more secure, but they can result in the rejection of many unlisted valid inputs.

OWASP has issued rules that define proper escaping schemes for inputs referenced in different HTML output locations.[1] For example, all three vulnerable statements in Figure 1 can be secured by applying proper escaping methods to the input variables at proper places, as in the following (escape() is the JavaScript library function that could encode HTML entities):

Table 1. Comparison of XSS defenses.

Method	Code modification	User involvement	Applicable before deployment	Generate concrete attack	Locate vulnerability	Input source ID	Runtime overhead	XSS exploits addressed
Defensive coding	Yes	Intensive	Yes	Not applicable	Not applicable	Not applicable	Not applicable	All types
Input validation testing	No	Intensive	Yes	Yes	Not explicitly	Yes	No	All types
Fault-based XSS testing	Yes	Intensive	Yes	Yes	Yes	Yes	No	All types
Static analysis	No	Average	Yes	No	Yes	Yes	No	Reflected and stored
Static string analysis	No	Low	Yes	Not explicitly	Yes	Yes	No	Reflected and stored
Combined static and dynamic analysis	No	Low	Yes	Yes	Yes	Yes	No	Reflected and stored
Server-side prevention	Yes	Average	No	No	No	No	Yes	All types
Client-side prevention	No	Intensive	No	No	No	No	Yes	All types

```
12 out.println("Your Message:  '"+
       HTMLencode(new_tip)+ "' is too long!");
18 out.println("'"+HTMLencode(tip)+"'");
25 document.write(escape(document.URL.
      substring(pos,document.URL.length)));
```

Defensive coding practices, if applied appropriately, can completely remove all XSS vulnerabilities in Web applications. However, they are labor-intensive, prone to human error, and difficult to enforce in deployed applications.

XSS testing

Input validation testing could uncover XSS vulnerabilities in Web applications. Specification-based IVT methods generate test cases with the aim of exercising various combinations of valid/invalid input conditions stated in specifications.[3] To avoid the sole dependency on specifications, Nuo Li and colleagues attempted to infer valid input conditions by analyzing input fields and their surrounding texts in client-side scripts.[3] Code-based IVT methods apply static analysis to extract valid/invalid input conditions from server-side scripts.[3] In general, the effectiveness of both specification- and code-based approaches relies largely on the completeness of specifications or the adequacy of generated test suites for discovering XSS vulnerabilities in source code.

Only test cases containing adequate XSS attack vectors can induce original and mutated programs to behave differently. Hossain Shahriar and Mohammad Zulkernine developed MUTEC, a *fault-based XSS testing* tool that creates mutated programs by changing sensitive program statements, or sinks, with mutation operators.[4] For example, for the sink at line 25 in Figure 1b, MUTEC, through its mutation operator ADES (Add escape function calls), generates

```
document.write(escape(document.URL.
      substring(pos,document.URL.length)));
```

and then attempts to find a test case that results in a different number of HTML tags between the original statement and its mutated statement. One such test case is

```
User → <Script>alert('XSSed!')</Script>
```

MUTEC generates adequate test suites for exposing XSS vulnerabilities but requires intensive labor as the task of generating mutants is not automated.

Vulnerability detection

Other XSS defenses focus on identifying vulnerabilities in server-side scripts. Static-analysis-based approaches can prove the absence of vulnerabilities, but they tend to generate many false positives. Recent approaches combine static analysis with dynamic analysis techniques to improve accuracy.

Static analysis. These techniques identify tainted inputs accessed from external data sources, track the flow of tainted data, and check if any reached sinks such as SQL statements and HTML output statements. Benjamin Livshits and Monica Lam used binary decision diagrams to apply points-to analysis to server-side scripts; their approach requires users to specify vulnerability patterns in Program Query Language.[5] Yichen Xie and Alex Aiken proposed a static analysis technique that obtains block and function summary information from symbolic execution.[6]

Pixy, an open source vulnerability scanner, includes alias analysis to improve accuracy.[7] For example, for the program travelerTip.jsp in Figure 1a, it reports the following statements as vulnerable:

```
11 out.println("Your Post has been added
       under Place '" + HTMLencode(place)+"'");
12 out.println("Your Message:  '"+
       new_tip+ "' is too long!");
18 out.println("'"+tip+"'");
```

In this case, the reported vulnerable statement at line 11 is a false positive because an escaping method sufficiently sanitizes the input. On the other hand, as it does not analyze travelerInfo.js in Figure 1b, it will miss a real vulnerable statement at line 25.

Static-analysis-based techniques quickly detect potential XSS vulnerabilities in source code and are relatively easy for security personnel to implement and adopt. However, they cannot check the correctness of input sanitization functions and, instead, generally assume that unhandled or unknown functions return unsafe data. These approaches also miss DOM-based XSS vulnerabilities as they do not target client-side scripts.

Static string analysis. Gary Wassermann and Zhendong Su enhanced the original taint-based approaches with string analysis.[8] Their technique uses context-free grammars (CFGs) to represent the values a string variable can hold at a certain program point, which facilitates the checking of blacklisted string values in sensitive program statements.

The enhancement provides more accuracy as it can analyze string operations' effects on inputs. However, when conducting static string analysis, it is difficult to model complex operations such as string-numeric interaction; thus, this approach can result in false positives if analysts make conservative approximations when handling such operations. Static string analysis also suffers from the limitations of blacklist comparisons.

For the program in Figure 1a, Wassermann and Su's approach produces the following CFGs (represented using Perl regular expression notation) for tainted strings at each sink:

```
At line 11: HTMLencode(place) →
                  ([^&<>]*(&(amp|lt|gt);)*[^&<>]*)*
At line 12: new_tip → .*
At line 18: tip → .*
```

In this case, static string analysis reports the statement at line 11 as safe because the expression ([^&<>]*(&(amp|lt|gt);)*[^&<>]*)* does not allow the tags < and >, which are the special characters an XSS exploit would use. It also reports that the statements at line 12 and 18 are unsafe because the expression .* represents any string values and, as such, any attack strings in new_tip and tip would execute at these statements.

Note that the program's control flow structure (line 9) dictates that the variable new_tip must contain at least 100 characters, but the CFG for new_tip results in the expression .* because Wassermann and Su's approach cannot handle string-numeric interactions.

Combined static and dynamic analysis. Motivated by static-analysis-based approaches' inability to identify faulty sanitization functions, Davide Balzarotti and colleagues developed the Saner tool, which checks the adequacy of sanitization functions for defending against XSS attacks.[7] This successor to Pixy uses a static string analysis method similar to that proposed by Wassermann and Su to first identify the potentially faulty sanitization methods, then simulates the identified methods with a set of test inputs that contain attack strings and checks if any attack could still reach the sinks.

Static-analysis-based approaches can prove the absence of XSS vulnerabilities, but they tend to generate many false positives.

Lam and colleagues carried out points-to analysis to track the flow of tainted data in a program and then used this information to instrument the program for model-checking purposes.[5] Applying the QED model checker based on Java Pathfinder (http://babelfish.arc.nasa.gov/trac/jpf), they simulated the instrumented program with inputs likely to lead to a match with user-specified vulnerability patterns. This approach's effectiveness depends on the completeness of the vulnerability specifications and QED's ability to explore as many different paths as possible.

Building on the work by Wassermann and colleagues, a team led by Adam Kiezun used *concolic* (concrete+symbolic) *execution* to capture program path constraints and a constraint solver to generate test inputs that explored various program paths.[9] Upon reaching the sinks, they exercised two sets of inputs—one of ordinary valid strings and the other of attack strings from a library (http://ha.ckers.org/xss.html)—and checked the differences between the resulting program behaviors.

The following concolic execution sequences lead to the generation of an attack string that exploits the vulnerable statement at line 12 in Figure 1a:

1. Assign a program's input parameters null values:

   ```
   action → null, place → null, new_tip → null
   ```

2. The program executes on these inputs, capturing the constraint

   ```
   !(place !=null && action.equals("Post"))
   ```

3. To explore a new control flow path, negate the captured constraint:

```
place !=null && action.equals("Post")
```

4. The constraint solver solves the above constraint and generates new input values:

```
action → Post, place → 1, new_tip → null
```

5. The program executes on these new inputs, capturing the constraint

```
place !=null && action.equals("Post")
     && new_tip.length < 100
```

Combined static and dynamic analysis can create concrete attack vectors and thus avoid false positives; it also enables fully automated test case generation.

6. Similar to (2), generate a new constraint:

```
place !=null && action.equals("Post")
     && new_tip.length >= 100
```

Kiezun and colleagues' current solver cannot solve the constraint new_tip.length >= 100 as it requires complex analysis of string operations that return numeric values. Instead of solving it, the concolic engine executes random inputs and checks if the path satisfying this constraint is exercised. If it generates the inputs

```
action → Post, place → 1,
new_tip → 1…1 (a hundred of '1's)
```

within a given time, the engine exercises a new control flow path and finds a sink referencing the tainted variable new_tip (at line 12).

Once it encounters such a statement, the attack generator alters the current values of the input parameters referenced in the statement using attack strings and re-executes the program. If the altered inputs result in the same control flow path, the engine finds a real attack vector such as

```
action → Post, place → 1,
new_tip → 1…1<Script>alert('XSSed!')</Script>
```

Combined static and dynamic analysis can create concrete attack vectors and thus avoid false positives; it also enables fully automated test case generation. However, current implementations only work on server-side scripts; more research is needed on client-side script analysis.

This approach has two other weaknesses, both of which can result in false negatives. First, the attack string library might not be complete due to the everyday introduction of new attacks. The program's sanitization routines might trap the sample attack vectors generated during testing, but some real-life attack vectors might circumvent those routines. Second, this technique suffers from state space explosion and thus might miss some vulnerabilities in deep state spaces.

Currently, there is no universal solution to the state space explosion problem. Among recent empirical studies, QED struggled at testing the Java-based JGossip (80,000 lines of code) Web application due to the more than 30 billion possible test cases, and Kiezun and colleagues' Ardilla tool only achieved 14 percent line coverage for the PHP-based phpBB (35,000 LOC) Web application.[5,9]

Characterizing bug patterns using pattern analysis and predicting unknown bugs using pattern matching and data mining might ease the first problem, while more effective string constraint solving techniques and AI algorithms could mitigate the second.

Runtime attack prevention

The final group of XSS defenses focus on preventing real-time attacks using intrusion detection systems or runtime monitors, which can be deployed on either the server side or client side. In general, these methods set up a proxy between the client and server to intercept incoming or outgoing HTTP traffic. The proxy then checks the HTTP data for illegal scripts or verifies the resulting URL connections against security policies.

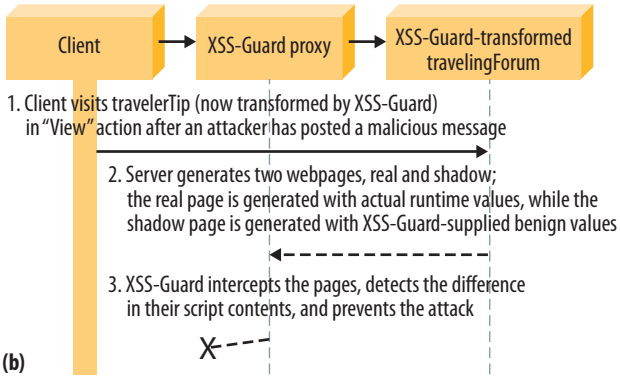
Server-side prevention. Yao-Wen Huang and colleagues developed the WebSSARI (Web Security via Static Analysis and Runtime Inspection) tool, which performs type-based static analysis to identify potentially vulnerable code sections and instrument them with runtime guards.[10] Users specify preconditions of sensitive functions—for example, those that contain HTML outputs—and postconditions of sanitization functions. During runtime, instrumented guards check for conformance of these user-specified conditions.

Other approaches use dynamic taint-tracking mechanisms to monitor the flow of input data at runtime.[10] They ensure that these inputs are syntactically confined (only treated as literal values) and do not contain unsafe content defined in user-specified security policies.

Instead of tracking tainted data, a proposed technique tracks untainted or trusted strings, such as those defined by programmers, and ensures that SQL statements' syntactic contents have only these strings.[10] Although this defense focuses on SQL injection, it can be extended to

```
StringBuffer re= “”; //real response
StringBuffer sh= “”;//shadow response
       . . .
re.append(“Here are the tips….”);
re.append(“Here are the tips….”);
while(rs.next()) {
     String tip= rs.getString(“tip”);
     String tip_c= “a”;
     re.append(“‘”+tip+“’”);
     sh.append(“‘”+tip_c+“’”);
       . . .
re.append(“</body></html>”);
sh.append(“</body></html>”);
re= XSS-PREVENT(re, sh);
```

(a)



(b)

Figure 4. XSS-Guard server-side prevention mechanism. (a) Sample code transformed from a part (lines 15-19) of the program in Figure 1a. (b) Server-side attack prevention.

address XSS because of the similarity of SQL injection and XSS exploits.

XSS-Guard transforms server programs so that they produce a shadow webpage, which reflects the program's intended output, for each real response page.[11] As Figure 4 shows, before sending the real response page to clients, XSS-Guard checks for differences in the script contents of real and shadow pages.

Some server-side prevention approaches require the collaboration of browsers. One example is BEEP (Browser-Enforced Embedded Policies), a mechanism that modifies the browser so that it cannot execute illegitimate scripts.[11] Security policies dictate what data the server sends to BEEP-enabled-browsers.

In contrast to BEEP, Blueprint is a server-side tool that works on existing browsers.[11] To evade unreliable parsing behaviors, the server takes over the browser's task of parsing untrusted HTML contents—those parts that might contain XSS vulnerabilities—and embeds the generated parse tree as a model in place of the actual untrusted content. It also embeds a model interpreter so that the browser can interpret the embedded models and produce the HTML documents as intended.

Server-side prevention can, in principle, prevent all XSS attacks because it checks actual runtime values of inputs and no approximation is necessary. However, it incurs runtime overhead due to interception of HTTP traffic. It also requires code instrumentation to enable dynamic monitoring and installation of additional (possibly complex) frameworks and, in some cases, user-defined security policies, both of which can be labor-intensive.

Client-side prevention. Noxes acts as a personal firewall that allows or blocks connections to websites on the basis of filter rules, which are basically user-specified URL whitelists and blacklists.[12] When the browser sends an HTTP request to an unknown website, Noxes immediately alerts the client, who chooses to permit or deny the connection, and remembers the client's action for future use.

Client-side prevention provides a personal protection layer for clients so that they need not rely on the security of Web applications. Its main disadvantage is that it requires client actions whenever a connection violates the filter rules. Moreover, although this approach addresses all types of XSS attacks, it only detects exploits that send user information to a third-party server, not other exploits such as those involving Web content manipulation.

TOOL SUPPORT

Implementations of some XSS defenses are available online.

To help developers practice its defensive coding rules, OWASP has created the Enterprise Security API (ESAPI; https://owasp.org/index.php/Category:OWASP_Enterprise_Security_API), an open source library for many different programming languages. Microsoft also provides the Web Protection Library (http://wpl.codeplex.com) for .NET developers. These libraries provide many escaping APIs and other security control features.

Pixy (http://pixybox.seclab.tuwien.ac.at/pixy) implements static analysis on PHP 4 source code. The string analyzer tool (http://score.is.tsukuba.ac.jp/~minamide/phpsa) also works on PHP programs. The concolic engine Ardilla is not available, but its underlying string constraint solver Hampi (http://people.csail.mit.edu/akiezun/hampi) is accessible as an independent tool. WebSSARI has been commercialized as CodeSecure (www.armorize.com). Noxes will soon be available as an open source tool (http://iseclab.org/projects/noxes). Currently it only works on MS Windows, and in a similar way to Windows personal firewalls.

Various off-the-shelf scanners can also detect XSS vulnerabilities. SecTools maintains a list of the top scanners (http://sectools.org/web-scanners.html). The list includes popular commercial systems such as Acunetix's Web Vulnerability Scanner (www.acunetix.com/vulnerability-scanner) and IBM's Rational AppScan family (http://

www-01.ibm.com/software/awdtools/appscan), as well as open source scanners such as OWASP's WebScarab (https://owasp.org/index.php/Category:OWASP_WebScarab_Project) and Paros (http://parosproxy.org). All of these scanners generally use either crawlers or proxies to fetch webpages and then inject predefined attack vectors into response pages, letting users verify the resulting behaviors.

Existing techniques for defending against XSS exploits suffer from various weaknesses: inherent limitations, incomplete implementations, complex frameworks, runtime overhead, and intensive manual-work requirements. Security researchers can address these weaknesses from two different perspectives.

From a development perspective, researchers need to craft simpler, better, and more flexible security defenses. They need to look beyond current techniques by incorporating more effective input validation and sanitization features. In time, development tools will incorporate security frameworks such as ESAPI that implement state-of-the-art technology.

From a program verification perspective, researchers must integrate program analysis, pattern recognition, concolic testing, data mining, and AI algorithms—heretofore used to solve different software engineering problems—to enhance the effectiveness of vulnerability detection. They can also improve the precision of current methods by acquiring attack code patterns from outside experts as soon as they become available.

References

1. Open Web Application Security Project, XSS (Cross-Site Scripting), Prevention Cheat Sheet, 2011; https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.
2. S. Fogie et al., *XSS Attacks: Cross Site Scripting Exploits and Defense*, Syngress, 2007.
3. N. Li et al., "Perturbation-Based User-Input-Validation Testing of Web Applications," *J. Systems and Software*, Nov. 2010, pp. 2263-2274.
4. H. Shahriar and M. Zulkernine, "MUTEC: Mutation-Based Testing of Cross Site Scripting," *Proc. 5th Int'l Workshop Software Eng. for Secure Systems* (SESS 09), IEEE, 2009, pp. 47-53.
5. M.S. Lam et al., "Securing Web Applications with Static and Dynamic Information Flow Tracking," *Proc. 2008 ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation* (PEPM 08), ACM, 2008, pp. 3-12.
6. Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," *Proc. 15th Usenix Security Symp.* (Usenix-SS 06), vol. 15, Usenix, 2006, pp. 179-192.
7. D. Balzarotti et al., "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," *Proc. 29th IEEE Symp. Security and Privacy* (SP 08), IEEE CS, 2008, pp. 387-401.
8. G. Wassermann and Z. Su, "Static Detection of Cross-Site Scripting Vulnerabilities," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), ACM, 2008, pp. 171-180.
9. A. Kiezun et al., "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS, 2009, pp. 199-209.
10. W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *IEEE Trans. Software Eng.*, Jan. 2008, pp. 65-81.
11. M.T. Louw and V.N. Venkatakrishnan, "Blueprint: Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers," *Proc. 30th IEEE Symp. Security and Privacy* (SP 09), IEEE CS, 2009, pp. 331-346.
12. E. Kirda et al., "Client-Side Cross-Site Scripting Protection," *Computers & Security*, Oct. 2009, pp. 592-604.

Lwin Khin Shar *is a research student in the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His research interests include software security and Web security. Shar received a BE in electrical and electronic engineering from Nanyang Technological University. Contact him at shar0035@ntu.edu.sg.*

Hee Beng Kuan Tan *is an associate professor of information engineering in the School of Electrical and Electronic Engineering, Nanyang Technological University. His research focuses on software security, analysis, and testing. Tan received a PhD in computer science from the National University of Singapore. He is a senior member of IEEE and a member of ACM. Contact him at ibktan@ntu.edu.sg.*